Docs

Kernel Project

Lightweight Kernel Implementation with DI Container

Anonymous · 15/11/2026 · #php

Table of Contents

Kernel Project 3
- ***Introduction*** 4
- ***Configuration*** 7
- ***Environment*** 10

Kernel Project

Lightweight Kernel Implementation with DI Container

Anonymous · 15/11/2026 · #php

Lightweight Kernel Implementation with DI Container

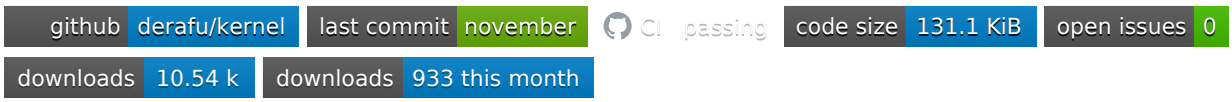
Last updated on 15/11/2026

Introduction

Lightweight kernel with a dependency injection container, inspired by Symfony, with .env files and a container cache

Anonymous · 15/11/2026

Lightweight Kernel Implementation with DI Container



A lightweight kernel implementation with a dependency injection container, inspired by Symfony but with a minimal approach.

Overview

Derafu Kernel provides a clean, flexible foundation for PHP applications with minimal dependencies. It offers:

- A lightweight kernel implementation with DI container.
- Environment-aware configuration management.
- Support for PHP and YAML configuration files.
- Container caching for improved performance.
- Built-in support for multiple environments (dev, prod, test, etc.).

Installation

```
composer require derafu/kernel
```

Basic Usage

Create a Kernel

The simplest way to use the kernel is to create an instance with a specific environment:

```
use Derafu\Kernel\MicroKernel;
```

```
// Create a kernel with 'dev' environment and debug mode enabled.
$kernel = new MicroKernel('dev', true);

// Boot the kernel to initialize the container.
$kernel->boot();
```

Custom Kernel Implementation

You can extend the `MicroKernel` class to customize its behavior:

```
use Derafu\Kernel\MicroKernel;
use Symfony\Component\DependencyInjection\Loader\Configurator\ContainerConfigurator;

class AppKernel extends MicroKernel
{
    // Override the default configuration files.
    protected const CONFIG_FILES = [
        'services.php' => 'php',
        'routes.php' => 'routes',
        'parameters.yaml' => 'yaml',
    ];

    // Add additional container configuration.
    protected function configure(ContainerConfigurator $configurator): void
    {
        $services = $configurator->services();

        // Register application services.
        $services->set('app.service', MyService::class)
            ->public()
            ->args(['param1', 'param2']);
    }
}
```

Environment Configuration

The kernel uses an environment object to determine settings and directories:

```
use Derafu\Kernel\Environment;

// Create an environment with custom settings.
$environment = new Environment(
    'prod',                 // Environment name.
    false,                  // Debug mode.
    ['custom' => 'value']   // Context variables.
);

// Create a kernel with the custom environment.
```

```
$kernel = new MicroKernel($environment);
```

Last updated on 15/11/2026

Configuration

Configuration

Anonymous · 15/11/2026

Configuration

Directory Structure

The kernel expects a `config` directory for configurations. Any other structure is free.

The directories come from `Environment`:

Directory	Where it is
Project	The `project` directory if it is given to `Environment`; otherwise the `PROJECT_DIR` of the context (what the application receives from the runtime); otherwise the root package of Composer, the project that is running. It is not deduced from where a package of the dependencies is installed.
Cache	`<project>/var/cache/<environment>`.
Configuration	The `config` directory of the project.
Logs	`<project>/var/log`.

The configuration files (`services.yaml`, `routes.yaml`...) are read **from the configuration directory only**. A file with the same name in the directory where the process runs is not read, whatever the working directory is.

Service Configuration

Define services in `config/services.php`:

```
use Symfony\Component\DependencyInjection\Loader\Configurator\ContainerConfigurator;

return static function (ContainerConfigurator $configurator) {
    $services = $configurator->services();

    // Auto-configure and autowire services by default.
    $services->defaults()
        ->autowire()
        ->autoconfigure();
```

```
    // Register a single service.
    $services->set('app.service', AppService::class)
        ->public();

    // Register multiple services from namespace.
    $services->load('App\\Service\\', '../src/Service/*')
        ->public();
};
```

Route Configuration

Define routes in `config/routes.php` or `config/routes.yaml`:

```
// routes.php
return [
    'home' => [
        'path' => '/',
        'controller' => 'App\\Controller\\HomeController::index',
    ],
    'blog_show' => [
        'path' => '/blog/{slug}',
        'controller' => 'App\\Controller\\BlogController::show',
        'parameters' => [
            'requirements' => [
                'slug' => '[a-z0-9-]+',
            ],
        ],
    ],
];
```

Or in YAML:

```
# routes.yaml
home:
    path: /
    controller: App\Controller\HomeController::index

blog_show:
    path: /blog/{slug}
    controller: App\Controller\BlogController::show
    parameters:
        requirements:
            slug: '[a-z0-9-]+'
```

The routes of `routes.yaml` and the ones of `routes.php` are **added**: each file is loaded after the one before it, so the routes of all of them are kept, in the order the files were loaded. A name that two files define has the route of the later one (it keeps its place in the order). The `imports` of a YAML file

are added the same way.

The Cache of the Container

The container is built from the configuration and dumped to a PHP file in the cache directory (`var/cache/<environment>/container_<id>.php`, where the id is the hash of the class of the kernel). The next boot loads that file instead of building the container again.

- **In debug mode** the container is built and dumped on every boot, so a change in the configuration is seen right away.
- **Without debug** (production) the file that is there is used as it is. **It is never checked against the configuration**: what the container was built with stays until the file is deleted. That includes `services.yaml` and `routes.yaml` (also the ones that the packages bring, so a `composer update` that changes them), the classes that a resource glob such as `App\Controller\:` finds, and the environment variables, because they are dumped as `env.<NAME>` parameters.

The kernel does not clear the cache: it is the deployment that has to start with an empty cache directory. The usual way is to not share `var/cache` between deploys (each release has its own), so every deploy builds the container once, with the configuration and the environment of that release. If the cache directory is shared or kept, delete `var/cache/<environment>` after changing the configuration, the `.env` or the dependencies.

Two kernels of the same class in the same process (for example in tests) do not share a container: the second one that is built gets its own, with the configuration of its moment. The cache file always declares the class with the name of the kernel, so the next process can load it.

Last updated on 15/11/2026

Environment

Environment

Anonymous · 15/11/2026

Environment

Environment Variables and the `.env` Files

The variables of the application come from the real environment (the process, the web server) and from the `.env` files of the project. `Environment` reads the files **when it is created**, before the container is built and before the kernel boots, from the project directory (the one with the `vendor/` of Composer).

The files

They are loaded in this order, by the name of the environment of the kernel (`dev`, `prod`, `test`...), and **the last one that has a variable wins**:

Order	File	Use
1	`.env`	The defaults of the project. Committed.
2	`.env.<environment>`	The values of one environment (`.env.prod`, `.env.test`...). Only the one of the environment of the kernel is read: `prod` never reads `.env.dev`.
3	`.env.<environment>.local`	The values of one environment on this machine. Not committed.
4	`.env.local`	The values of this machine, for every environment. Not committed.

- **In `test` the `.local` files are not loaded** (3 and 4), so the machine of whoever runs the tests does not change their result.
- A file that does not exist is skipped. Nothing is read from `.env.dist` or similar names.
- **A variable of the real environment is never replaced by a file**, but see the note below about `variables_order`.
- A variable that two files define has the value of the later one (`.env.local` over `.env`).

Only the files in the table are loaded. The environment that decides which ones is the **name of the kernel's environment**, not the `APP_ENV` variable.

`APP_ENV`

The kernel does not define `APP_ENV`: if nobody defines it, it stays undefined. The name of the environment of a web application is decided before the files are read: `Derafu\Http\Runtime::getApplicationContext()` takes `APP_ENV` from `$_SERVER` and `$_ENV` (the real environment) and uses `dev` when it is not there, and that is the name that `new Environment($context['APP_ENV'], ...)` receives. So `APP_ENV=prod` written in a `.env` does not make the kernel `prod`: define it in the real environment (the web server or the container).

What the kernel does with them

- **Parameters:** every variable that is in `$_ENV` after reading the files becomes the parameter `env.<NAME>`. They are dumped to the cache of the container with it, so a change in a `.env` is not seen in production until the cache is cleared.
- **`%env()%` in the configuration:** `%env(NAME)%` and the typed forms (`%env(bool:NAME)%`, `int`, `float`, `json`, `string`, `default::`…) are read **when a service that uses them is created**, from `$_ENV`, `$_SERVER` and the process. That can be long after boot, when a page needs the service for the first time.
- **`Environment::getEnv($name, $default)`:** the same, with an optional type (`getEnv('bool:APP_DEBUG')`).
- **The super-globals:** reading the files writes the variables to `$_ENV` and `$_SERVER`. A test that checks the global state has to build the kernel before it takes its photo (see testing a site).

`variables_order`

The files do not replace the variables that are in `$_ENV` or `$_SERVER`, and those are the only places that are looked at. If the `variables_order` of PHP has no `E` (for example `GPCS`), `$_ENV` is empty, and a variable that exists only in the process environment (`getenv()`) is not seen: the value of the file is used. Under a web server the variables usually are in `$_SERVER`, so the real value wins; on the command line it depends on the configuration of PHP.

Environment Types

The kernel supports multiple environment types through constants in `EnvironmentInterface`:

- `LOCAL`: Local development environment.
- `DEVELOPMENT`: Development environment.
- `TEST`: Testing environment.
- `STAGING`: Staging environment.
- `QUALITY_ASSURANCE`: QA environment.
- `PREPRODUCTION`: Pre-production environment.
- `PRODUCTION`: Production environment.